Re-implementing a Ruby on Rails Site in Yesod

CS 557: Functional Languages @ Portland State University
Dylan Laufenberg, winter 2019

1 Introduction

For the last seven years or so, I've been developing and expanding a Ruby on Rails-powered website for a Sacramento, CA soup kitchen named Sharing God's Bounty. I've had mixed feelings about my choice of Ruby on Rails over the last half-dozen years, so this final project provides an excellent opportunity to experiment with a significantly different Web technology stack than any other that I've used. I implement a narrow, vertical slice of functionality to experience and experiment with Haskell across many aspects of server-side code for the Web. I use the Yesod Web framework for my project, and I provide comparisons to my Ruby on Rails experience as appropriate throughout this report. I followethe quick start instructions here, which have provided me with Yesod 1.6.0.3.

Yesod's documentation comes primarily from the Yesod book [1] (available online here), so a large portion of my work for this project has simply been reading through the book to understand how to utilize this highly sophisticated, mature framework. I don't attempt to introduce or define every concept in Yesod here. Instead, I highlight particularly noteworthy applications of Haskell to the domain of Web application development and discuss how I use Yesod to build out the features I develop for this final project. In particular, I assume the reader knows about Haskell features such as Template Haskell, QuasiQuotes, and language pragmas either through prior knowledge or by reading the Haskell section of the Yesod book [2].

New section: Cover "The Basics"? Regardless, need some kind of transition.

2 Resources and type-safe URIs

One of the first concepts we need to discuss in Yesod, and one of the clearest examples of applying Haskell type safety to an unexpected domain, is the problem of constructing a routing system to bind certain URIs to associated server-side actions.

The most fundamental term here, one which Yesod shares with many other Web frameworks, is the **route**, which is a mapping from a URI to server-side code. To represent routes, Yesod uses what it calls resources. A **resource** is a `data` constructor that acts as the canonical name of the route in views [1]. For instance, the default site scaffold includes a static route of the form:

```
/static StaticR Static appStatic
```

This line specifies a `/static` route, a resource constructor named `StaticR` as the name of the route, and a subsite named `Static` whose function `appStatic` will handle requests on this route [3]. In order to create a link to a static file, e.g. an image served as part of a template, we invoke the `StaticR` constructor with an appropriate argument. The `Static` subsite generates "static file identifiers" at compile time [4]. When we insert a link using a resource constructor:

```
<img ... src="@{StaticR img_logo_png}" ... />
```

Yesod generates a correct link to the static resource at compile-time. In this case, the user's browser receives:

```
<img ... src="http://localhost:3000/static/img/logo.png?etag=GVVUzjtL" ... />
```

2.1 Benefits of Resources

There are two primary benefits to this approach, both of which leverage Haskell features from class (albeit at a very advanced level).

Check links at compile time Yesod uses resources to generate correct links at compile-time, which allows it to check whether said links are valid and raise compile-time errors if any problems are detected. In this case, the Static subisite checks the identifier `img_logo_png` against all generated static file identifiers. If it finds a match, it inserts the correct link. If not, then GHC raises an error. For instance, if we mistakenly specify a JPG logo:

```
<img ... src="@{StaticR img_logo_jpg}" ... />
```

GHC detects the issue for us and (rather obtusely) notifies us of the issue with a compile-time error:

```
    • Variable not in scope: img_logo_jpg :: Route Static
    • Perhaps you meant ‘img_logo_png’ (imported from Import.NoFoundation)
    |
163 |             $(widgetFile "default-layout")
    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In effect, Yesod automatically tests every internal link at compile time, without the need to manually write tests against the generated HTML. We, the developers, need to test our functions for generating links, then Yesod takes care of the rest for us. In this case, I feel comfortable as a developer taking it on faith that the static file subsite is well-tested.

Refactor routes with confidence Generating links at compile time also eliminates an entire class of common and nefarious errors: broken links. Updating or moving routes is a common occurrence in Web application development, even with careful planning. Yesod makes this safe and automatic by generating all such links from resource constructors at compile time. To change the route across the entire application, all we need to do is change the route definition:

```
/dynamic StaticR Static appStatic
```

And all of our generated links to that route automatically update when we recompile:

```
<img ... src="http://localhost:3000/dynamic/img/logo.png?etag=GVVUzjtL" ... />
```

3 Shakespearean Templates

Yesod provides domain-specific languages (DSLs) over HTML, CSS, and Javascript, all named after Shakespearean characters. In converting my site, I migrate the HTML, CSS, and Javascript, in that order. (For the full, reference versions of all information in this section that's not otherwise cited, see [5].)

All of Yesod's DSLs provide a common set of interpolation features:

- **Variable interpolation** All values in scope when a template is rendered are available through variable interpolation. For instance, in my main site template, I write

  ```
  <title>#{pageTitle pc} - Sharing God's Bounty
  ```

  to insert the value of `pageTitle pc` as part of the HTML title.

- **Resource interpolation** We can insert type-safe URIs via `@{routeR}` resource interpolation, as discussed in Resources and type-safe URIs.

- **Template interpolation** We can embed templates in other templates by writing `^{templateName}`. This is safe in that it only allows embedding of templates of the same type and, of course, provides all the usual compile-time guarantees for both templates. (See the Mixins section for more.)

3.1 Hamlet

Yesod uses Hamlet as its HTML DSL. Hamlet is similar to HTML, with a few distinguishing features.

Significant whitespace Hamlet templates infer closing tags from indentation levels. When you write, for instance, `<div>`, everything that should appear inside this tag must be nested below it. A matching closing tag is automatically inserted at the correct place in the generated HTML.

Class and ID shorthand Hamlet templates provide shorthand for specifying classes and IDs by prefacing a class or ID name with a `.` or a `#`, respectively. This shorthand even works with multiple classes, by writing either `<div .foo .bar>` or `<div ."foo bar">`. Either form generates the final HTML `<div class="foo bar"></div>`. To be honest, I tried these forms with the intention of exposing a shortcoming of Hamlet, and I was shocked to see that it handled both cases beautifully!

Example To see these features in action, let's take a look at my mobile navigation menu. The Hamlet representation is as follows: **UPDATE THIS EXAMPLE with the "final" version! It'll be WAY more impressive!**

```
<nav .mobile>
  <ul>
    <li>Menu items go here.
  <h2>
    Menu
```

Which compiles to:

```
<nav class="mobile"><ul><li>Menu items go here. </li>
</ul>
<h2>Menu</h2>
</nav>
```

Shortcomings There are, unfortunately, a number of shortcomings to Hamlet. First and perhaps most importantly, none of the four templating DSLs provide the strong, compile-time guarantees of correctness that we expect in Haskell, even though Hamlet in particular feels like it could. As a new user, I was genuinely surprised to learn that Hamlet doesn't raise compile-time errors for invalid tags! For instance, I once mistakenly wrote `<image ...>`, and Hamlet happily generated both opening and closing `image` tags, which, understandably, confused both Firefox and me.

Second, although Hamlet knows that some tags like `img` are self-closing, it doesn't insert the closing `/` at the ends of such tags.

Third, Hamlet *does not* check whether classes and IDs written in Hamlet notation exist in linked Cassius or Lucius files!

Fourth, the HTML produced seems to be both arbitrary and virtually unreadable in its structure. The navigation example above is Hamlet's output, verbatim, and this is with minimization disabled! What particularly irks me about this oversight is that Hamlet *outright requires* proper indentation and spacing! Why it doesn't simply carry this indentation through to the final HTML is beyond my comprehension.

3.2 Lucius

To represent CSS, Yesod uses two, equivalent DSLs: Lucius, which is "a superset of CSS" [5], and Cassius, which uses significant whitespace instead of curly braces. I chose to use Lucius to ease the process the migration from Ruby on Rails.

Lucius and Cassius both present broadly similar feature sets to Sass, which my Ruby on Rails app uses for its CSS. In fact, in searching for help on Lucius mixins, I found a question by a Sass user who was similarly struggling with the Lucius equivalent [10]. (More on that in the Mixins section.) Put simply, Lucius and Cassius aim to ease the process of writing CSS but *do not* provide strong compile-time guarantees of the correctness of CSS.

3.2.1 CSS variables

Lucius supports fairly standard features to simplify generating CSS. In addition to interpolation, it supports another, immensely useful feature: CSS variables. Other than the surprising (and unfriendly) syntactic detail that we write `@foo` to define a variable and `#{foo}` to use it, variables in Lucius feel familiar after working with Hamlet. With this understanding, I begin migrating my Sass by writing some Lucius variables at the top of my `.lucius` file:

```
@textcolor: #333;
@pageBGDark: #382916; /* Brown */
@pageTextLight: white;
@spacing: 20px;
@halfSpacing: 10px;
```

Here, `@spacing` and `@halfspacing` support the grid on which the front-end design is based and open up a new possibility that, should I ever wish to adjust this grid, I might well be able to do so simply by changing these variables. However, this potential is limited somewhat by an unfortunate limitation of Lucius: variables are treated as literal text and thus cannot invoke other variables! When we write, for instance:

```
@spacing: 20px;
@halfSpacing: calc(#{spacing}/2);

body {
  margin: 0 #{halfSpacing};
}
```

Lucius generates the CSS:

```
body {
  margin: 0 calc(#{spacing}/2);
}
```

If Lucius supported either nested variables or some notion of compile-time calculation with variables, I might be able to scale nearly every element on the page according to the `@spacing`. It's disappointing that Yesod ships with this limitation, but it downright disheartens me that Lucius does not even notice that it's generating obviously invalid CSS!

3.2.2 Nested scopes

Lucius also allows us to nest selectors, and it will unpack these selectors when it generates final CSS. For instance, my masthead CSS takes full advantage of this feature:

```
.masthead {
  background-color: #{pageBGDark};
  padding: #{spacing};
  text-align: center;

  box-shadow: 0 0 20px black;

  a img {
  display: block;
  ^{centering "200px"}
  }

  .logo {
    margin-bottom: #{spacing};
  }

  h2 {
    margin-top: 30px;
    margin-bottom: 0;
```

```
  }
}
```

Let's consider this example in detail. First, we see three variables being interpolated: `pageBGDark`, followed by `spacing` twice. (The shadow does not use `#{spacing}` variable because I choose to keep the shadows separate from the rest of the grid.) We also see template interpolation with `^{centering "200px"}`; we'll discuss this in the Mixins section. Of particular note, though, is that within the `.masthead` selector, we see both CSS properties *and additional selectors*. At compile time, Lucius unpacks these nested selectors, prepending `.masthead` to each one. This particular section compiles to the final CSS:

```
.masthead {
    background-color: #382916;
    padding: 20px;
    text-align: center;
    box-shadow: 0 0 20px black;
}
.masthead a img {
    display: block;
    width: 200px;
    margin-left: auto;
    margin-right: auto;
}
.masthead .logo {
    margin-bottom: 20px;
}
.masthead h2 {
    margin-top: 30px;
    margin-bottom: 0;
}
.masthead p {
    color: white;
    margin: 0;
}
```

3.2.3 Mixins

Lucius mixins — CSS-specific templates written using QuasiQuoters and included using template interpolation — are where I first experience the growing pains of learning Yesod. In my experience, when learning a new framework, there's almost always some, unforeseen hurdle that at least temporarily destroys my enthusiasm and for the first time makes me question my choice of framework. In Yesod, my first significant hurdle is the Lucius mixin system.

The Yesod book's example [5] appears simple enough:

```
-- Our mixin, which provides a number of vendor prefixes for transitions.
transition val =
    [luciusMixin|
        -webkit-transition: #{val};
        -moz-transition: #{val};
        -ms-transition: #{val};
        -o-transition: #{val};
        transition: #{val};
    |]
```

Essentially, this is simply a Lucius-specific QuasiQuoter that takes arbitrary text and uses it in a CSS block.

In the example in the Nested scopes section, the template interpolation `^{centering "200px"}` is defined as a mixin. It's very simple: it sets width according to its `width` parameter and sets left and right margins to `auto`, which is a standard way to center block-level HTML elements. Given that this is a common pattern in CSS, it's a good candidate for a Lucius mixin.

Following the Yesod book's example, I first try simply writing my QuasiQuoter in my Lucius file, right below my variable definitions. (After all, where else would my CSS go?) I write:

```
centering width =
  [luciusMixin|
    width: #{width};
    margin-left: auto;
    margin-right: auto;
  |]
```

Unfortunately, Yesod seems to be unaware of the QuasiQuoter. Since the mixin QuasiQuoter shares syntax with Lucius variable interpolation, and since I don't have a variable named `width` defined in my Lucius file, GHC produces a compilation error:

```
error: Variable not in scope: width
    |
163 |             $(widgetFile "default-layout")
    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

To confirm that this misunderstanding of variable interpolation is the cause of the compilation error and to confirm that Lucius mixins can't appear in Lucius files, we can simply remove the variable interpolation:

```
centering width =
  [luciusMixin|
    width: width;
    margin-left: auto;
    margin-right: auto;
  |]
```

With this modification, the Yesod application compiles, and it produces this same text, verbatim, in the generated CSS file. Clearly, something is amiss with our attempted use of mixins.

3.2.4 Troubleshooting mixins

Close inspection of the Yesod book's mixin example [5] reveals that it actually declares its mixin as a Haskell function that's in scope when the layout is rendered. With great hesitation, and in the name of learning Yesod, I consent to moving my mixin into Foundations.hs, from where the renderer is invoked.

In order to limit the function's scope appropriately, we follow the convention in the scaffolded file and use a `let` clause to declare the function locally within the `defaultLayout` widget **See widgets section for more details**:

```
defaultLayout :: Widget -> Handler Html
defaultLayout widget = do
  ...
  let centering width =
        [luciusMixin|
          width: #{width};
          margin-left: auto;
          margin-right: auto;
        |]
  ...
```

Unfortunately, just as the Yesod book warns [2], GHC complains of an ambiguous type for `width`. Specifically, Yesod uses the `OverloadedStrings` pragma to overload the `String` type such that either `String` or `Text` may represent a string, and in this case, GHC doesn't know which to use. Both class and the Yesod book mention the `ExtendedDefaultRules` pragma as a potential way to resolve this ambiguity. However, since both class and the Yesod book independently chasten us against relying on `ExtendedDefaultRules`, we instead search for an alternate, Dr. Jones-approved solution such as a type annotation.

The first challenge in creating a type annotation is determining what type the QuasiQuoter returns! Since QuasiQuoters are embedded Template Haskell, we brace ourselves for a complex, difficult-to-parse answer. We search Google and find no useful results. (This is left as an exercise for the reader.) We search Hoogle for "luciusMixin" and find no results. We check the Yesod wiki [7] and find that (a) it simply links to a Markdown file within a Yesod cookbook GitHub repository and (b) all its links appear to be broken! We pause briefly to catch our breath and let a wave of frustration pass.

Eventually, we find a Yesod blog post [8] that explains that, unfortunately, Yesod introduced Lucius mixins but did not support reloading mixins automatically when they changed during development until version 1.0.6.1. Noting that version number looks suspiciously like the Yesod 1.0.6.5 we are using through Stack, and that the blog post is dated July 1, 2013, we begin to question whether our version of Yesod even supports the syntax specified in the Yesod book! (Answer: it does; Shakespeare's version numbers don't appear to have any correlation to Yesod's version numbers.)

Eventually, we locate the `Text.Lucius` module [9] in the Shakespeare package with help from Hackage. We observe that `luciusMixin :: QuasiQuoter`, which seems unhelpful and unlikely to be correct (and is, indeed, wrong). We fire up GHCi and attempt to `import Text.Lucius`, but to no avail: either GHCi doesn't know about Stack, or Yesod's custom import system has foiled us.

A technique long honed by savvy Ruby on Rails developers (such as I) saves the day: if we simply provide an incorrect type annotation, perhaps GHC will tell us what it was expecting to see! A helpful StackOverflow answer [6] explains that by adding the `ScopedTypeVariables` language pragma, we can provide type annotations inside `let` statements. Thus, we ask Stack to run our application with our mixin defined as:

```
defaultLayout :: Widget -> Handler Html
defaultLayout widget = do
  ...
  -- Define Lucius mixins
  let centering :: Text -> Int
      centering width =
        [luciusMixin|
          width: #{width};
          margin-left: auto;
          margin-right: auto;
        |]
  ...
```

As desired and expected, GHC catches our incorrect type and very helpfully states:

```
    • Couldn't match expected type ‘Mixin’ with actual type ‘Int’
    • In the first argument of ‘Text.Internal.Css.CDMixin’, namely
        ‘(centering "200px")’
      In the expression: Text.Internal.Css.CDMixin (centering "200px")
      In the expression:
        ((Text.Shakespeare.Base.DerefBranch
            (Text.Shakespeare.Base.DerefIdent
               (Text.Shakespeare.Base.Ident "centering")))
          (Text.Shakespeare.Base.DerefString "200px"),
         Text.Internal.Css.CDMixin (centering "200px"))
    |
163 |             $(widgetFile "default-layout")
    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Eureka! We adjust our mixin accordingly:

```
defaultLayout :: Widget -> Handler Html
defaultLayout widget = do
  ...
  -- Define Lucius mixins
  let centering :: Text -> Mixin
      centering width =
        [luciusMixin|
```

```
          width: #{width};
          margin-left: auto;
          margin-right: auto;
        |]
...
```

And, at long last, the mixin compiles and works correctly, generating the code seen in Nested scopes!

3.3 Julius

According to the Yesod book, "Julius allows the three forms of interpolation we've mentioned so far, and otherwise applies no transformations to your content" [5]. Given how minimal it is and how little Javascript the Sharing God's Bounty website uses, we'll skip the subject of Julius entirely.

4 Reflections

Collect these thoughts into a real, formalized section in its own file.

Reflect on Yesod from the perspective of someone who's just made it through the basics. In particular, TinyOS's observation about being hard for beginners. I feel like I have to be an expert to do almost anything. Every piece of code feels like it's written for experts. And the lack of an accessible API reference (with good documentation) so far seems to be frustrating me.

Three points from TinyOS:

1. By making it harder to write bugs, you also make it harder to write code
2. By making hard things possible, you make easy things hard
3. By tailoring the language for expert users, you alienate newcomers and stifle your language's growth.

References

[1] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. 1st edition. O'Reilly, 2012. URL: https://www.yesodweb.com/book.

[2] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Haskell. URL: https://www.yesodweb.com/book/haskell.

[3] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Routing and Handlers. URL: https://www.yesodweb.com/book/routing-and-handlers.

[4] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Scaffolding and the Site Template. URL: https://www.yesodweb.com/book/scaffolding-and-the-site-template.

[5] Michael Snoyman. "Developing Web Applications with Haskell and Yesod". In: 1st edition. O'Reilly, 2012. Chap. Shakespearean Templates. URL: https://www.yesodweb.com/book/shakespearean-templates.

[6] *How to add type annotation in let binding*. URL: https://stackoverflow.com/questions/22436402/how-to-add-type-annotation-in-let-binding.

[7] psibi. *Home.md*. URL: https://github.com/yesodweb/yesod-cookbook/blob/master/cookbook/Home.md.

[8] Michael Snoyman. *Runtime Lucius: now with mixins!* URL: https://www.yesodweb.com/blog/2013/07/runtime-lucius-mixins.

[9] *Text.Lucius*. URL: http://hackage.haskell.org/package/shakespeare-2.0.20/docs/Text-Lucius.html.

[10] *Want to replace Sass with Lucius or Cassius but running into problems*. URL: https://stackoverflow.com/questions/21978823/want-to-replace-sass-with-lucius-or-cassius-but-running-into-problems.